Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Numerical Analysis and CUDA-Based Optimization of the Wave Equation

M.Sc. Dissertation in High-Performance Computing

School of Mathematics

Lishuang Sun

Supervised by Mike Peardon, Kirk M. Soodhalter

Trinity College Dublin

August 25, 2025

Declaration

I have read and understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at https://www.tcd.ie/calendar/.

I have also read and understood the guide, and completed the "Ready, Steady, Write" tutorial on avoiding plagiarism, located at https://libguides.tcd.ie/academic-integrity/ready-steady-write.

Name Surname

Abstract

Wave equations are core to physics and engineering, and find applications everywhere, in acoustics and seismology to electromagnetic wave propagation. As analytical solutions tend to only cover simple geometries with simple boundary conditions, numerical simulation is critical to solving realistic problems LeVeque (2007); Strang (2007).

The analysis is then generalized to a two dimensional (2D) wave equation in which the Leapfrog method is adopted with a five-point stencil applied to spatial discretization. We explore the related stability restrictions, explore numerical experiments such as standing wave , and evaluate error norms to make sure theoretical convergence rates.

The 1D and 2D schemes are run on a graphics processing unit (GPU) using CUDA by NVIDIA in order to address computational issues Sanders and Kandrot (2021). Techniques such as parallelization, 1D and 2D thread-block mappings and optimization of memory are used to effectively utilize full potential of GPU. Benchmark tests show large speedup over CPU implementations, particularly in the large-scale cases.

The numerical experiments verify the expected second-order accuracy and, in smooth starting conditions, the Leapfrog scheme is also superconvergent. These results identify the synergy between traditional numerical schemes and state-of-the-art high-performance computing (HPC), allowing the use of a broader and yet efficient framework to simulate wave equations.

The conclusion is that GPU-accelerated Leapfrog solvers are accurate and efficient with the possibility of extending the solver to higher dimensions. Future work can focus on absorbing boundary conditions, three-dimensional wave propagation and hybrid cross-approaches such as MPI+CUDA at large scales.

Acknowledgements

Thank you to everyone who helped me!

Contents

Chapter 1

Introduction

The wave equations form the basic mathematical description of the propagation of waves in many physical systems, including acoustics, seismology and electromagnetism. Their fundamental importance to both science and engineering is explained by the fact that they can capture necessary dynamical behaviour in a system where oscillations or disturbances propagate through space and time LeVeque (2007); Strang (2007). But analytical solutions to the wave equation are normally only available in simple geometries and boundary conditions. Numerical techniques are essential to carry out simulations in the real world since applications are usually quite complicated.

The finite difference schemes are one of the various ways to numerically solve the wave equation, however, it is evident that compared to the other approaches, the approach has advantages of being simple and having a relatively easy physical interpretation and that it is flexible to be used in high-performance computing (HPC) environments. In this category, explicit schemes will be most common in hyperbolic problems, leapfrog being an example. The Leapfrog scheme is second order and accurate on time and space, non-dissipative and computationally efficient yet bounded with respect to a Courant-Frederichs-Lewis (CFL) condition that controls its stability Ern and Steins (2020); Grote and Mitkova (2020). One limitation notwithstanding, it is well suited by its specificity to parallel architectures.

High performance computing has become a very important aspect in contemporary science. The emergence of graphics processing units (GPUs) means numeric simulations can be run at scale that previously was impossible on conventional CPUs. Massive parallelism is readily achieved using GPUs at relatively low cost, and a lineage of programming frameworks including CUDA have made it possible to effectively implement classical numerical methods to use the entirety of this hardware Sanders

and Kandrot (2021). GPU acceleration can produce orders of magnitude performance improvement over CPU-only implementations in wave propagation problems, which have a repetitive stencil computation over large grids.

Numerical solution to the wave equation with the Leapfrog scheme is explored in this dissertation with the attention being given to its application and efficiency in modern GPU-based computers. We consider initially the one-dimensional (1D) type, and a complete derivation of the scheme, the stability analysis, and convergence tests are provided. We then give the same treatment of the two-dimensional (2D) wave equation and modify the Leapfrog scheme by using a five-point stencil discretization and obtaining the similar stability conditions. Experiments through numerical experiments suggested the accuracy of the method and also the superconvergence nature at some conditions in both 1D and 2D. The CUDA-based implementations on GPU are greatly accelerated as seen using performance benchmarks.

This paper has three contributions:

- A derivation and full application of the Leapfrog method of the 1D and 2D wave equations, to include both a theoretical study of stability, accuracy.
- Implementations of the GPU-based solver based on CUDA showing significant speedup compared to computational processor solvers, with the observed scaling of the problem with regards to grid size.
- A convergence and error analysis which revealed that with smooth initial conditions, and appropriate CFL ratios the Leapfrog scheme can present superconvergent behaviour.

The remainder of this dissertation is organized as follows. Section 2 introduces the one- and two-dimensional mathematical models of the wave equation, together with the initial and boundary conditions. Section 3 presents the finite difference discretizations, with particular emphasis on the Leapfrog scheme. Section 4 describes the CUDA GPU implementation and optimization strategies. Section 5 provides the convergence and error analysis, along with numerical experiments and benchmark results. Finally, Section 6 concludes the work and outlines possible future directions, including extensions to three-dimensional problems and the use of hybrid MPI+CUDA approaches.

Chapter 2

Mathematical Model

A good approximation of the governing equations is the first step in designing and analyzing numerical methods for wave propagation. This chapter presents the continuous models, on which the following discretization and implementation are based. Our attention is concentrated on the classical linear wave equation that is a paradigmatic hyperbolic partial differential equation (PDE).

This discussion starts with the one-dimensional (1D) wave equation, that is of physical intuitive and analytic tractability. It is in the 1D case that we can present some basic notions like boundary conditions or the initial conditions and the exact solutions. We then generalize the formulation to the two-dimensional (2D) wave equation, which applies more realistically to practice in acoustics, seismology and electromagnetism. Simple settings may allow closed-form analytical solutions to both 1D and 2D equations, but these solutions are not known to exist under typical conditions in 2D, thereby providing the motivation to use numerical simulation and high-performance computing.

The models shown in this chapter are used to underpin the finite difference schemes shown in Section 3, as well as the GPU applications in Section 4.

2.1 One-Dimensional Wave Equation

The one-dimensional wave equation is a prototypical model that describes the transverse vibrations of an ideal string under tension. It is also widely applicable to acoustics, seismology, and other physical systems where disturbances propagate along a single spatial dimension. The governing partial differential equation (PDE) is given

by

$$\frac{\partial^2 u}{\partial t^2}(x,t) = c^2 \frac{\partial^2 u}{\partial x^2}(x,t), \quad x \in [0,L],\ t > 0, \tag{2.1}$$

where $u(x,t)$ denotes the displacement of the string at position x and time t, and c is the wave speed determined by the string's physical parameters:

$$c = \sqrt{\frac{T}{\rho}}, \tag{2.2}$$

with T the tension and ρ the linear mass density of the string.

Boundary conditions. To close the problem, appropriate boundary conditions must be imposed. For a string fixed at both ends, we enforce

$$u(0,t) = 0, \quad u(L,t) = 0, \quad t > 0, \tag{2.3}$$

which represent the fact that the ends of the string are clamped. Alternatively, Neumann-type conditions may be considered if the string is free at the boundary:

$$\frac{\partial u}{\partial x}(0,t) = 0, \quad \frac{\partial u}{\partial x}(L,t) = 0. \tag{2.4}$$

Initial conditions. The solution further requires initial displacement and velocity distributions:

$$u(x,0) = f(x), \quad \frac{\partial u}{\partial t}(x,0) = g(x), \quad x \in [0,L], \tag{2.5}$$

where $f(x)$ prescribes the initial shape of the string and $g(x)$ its initial velocity. A commonly studied example is a sine-shaped displacement with zero initial velocity:

$$u(x,0) = \sin\left(\frac{\pi x}{L}\right), \quad \frac{\partial u}{\partial t}(x,0) = 0, \tag{2.6}$$

which corresponds physically to a string plucked at its midpoint and released from rest.

Analytical solution. For the case of fixed-end boundaries and sine-shaped initial conditions, the problem admits a closed-form analytical solution via separation of variables:

$$u(x,t) = \sin\left(\frac{\pi x}{L}\right)\cos\left(\frac{\pi c t}{L}\right). \tag{2.7}$$

This solution oscillates periodically in time, representing the fundamental vibration mode of the string. It provides a useful reference against which the accuracy of numerical schemes can be tested.

Remarks. The one-dimensional wave equation illustrates key aspects of hyperbolic PDEs: finite propagation speed, energy conservation, and oscillatory dynamics. Although its analytical solution is tractable in simple cases, more complex initial or boundary conditions (e.g., nonuniform tension, absorbing boundaries) require numerical methods. In the next section, we extend this formulation to two spatial dimensions, where analytical solutions are rare and numerical discretizations become indispensable.

2.2 Two-Dimensional Wave Equation

Although the one-dimensional model represents the fundamental features of the wave propagation along with the same direction, there are numerous practical cases of the necessity to study the wave motion in the multiple dimensions. Specifically, two-dimensional (2D) waves, wave equation the equation can be used to describe the propagation of waves in two-dimensional domains in acoustics, seismology and electromagnetics. Examples are seismic wave propagation in the crust of the earth, acoustic imaging in medical imaging, electromagnetic wave scattering in radar, and so on.

Governing equation. The 2D linear wave equation is given by

$$\frac{\partial^2 u}{\partial t^2}(x,y,t) = c^2 \left(\frac{\partial^2 u}{\partial x^2}(x,y,t) + \frac{\partial^2 u}{\partial y^2}(x,y,t) \right), \quad (x,y) \in \Omega \subset \mathbb{R}^2,\ t > 0, \tag{2.8}$$

where $u(x,y,t)$ represents the displacement (or pressure, or field intensity, depending on the application), and c is the wave speed, assumed constant in this study.

Boundary conditions. The problem must be equipped with suitable boundary conditions. The most common types are:

- **Dirichlet (fixed boundary):**

$$u(x,y,t) = 0, \quad (x,y) \in \partial\Omega,\ t > 0.$$

  This models a rigid boundary where the wave field vanishes.

- **Neumann (free boundary):**

$$\frac{\partial u}{\partial n}(x,y,t) = 0, \quad (x,y) \in \partial\Omega,\ t > 0,$$

where $\partial u/\partial n$ denotes the outward normal derivative. This corresponds to a boundary where no flux crosses the domain boundary.

- **Absorbing (open boundary):** In unbounded domains, absorbing boundary conditions (or perfectly matched layers, PML) are often introduced to prevent artificial reflections. While such conditions are beyond the scope of the present implementation, they are essential for large-scale realistic simulations.

Initial conditions. As in the one-dimensional case, both displacement and velocity initial states must be prescribed:

$$u(x, y, 0) = f(x, y), \quad \frac{\partial u}{\partial t}(x, y, 0) = g(x, y), \quad (x, y) \in \Omega. \tag{2.9}$$

For instance, a commonly used test case is a standing wave:

$$u(x, y, 0) = \sin(k_x x) \cos(k_y y), \quad \frac{\partial u}{\partial t}(x, y, 0) = 0, \tag{2.10}$$

where k_x and k_y are the wave numbers in the x- and y-directions, respectively. This initial condition is smooth and separable, making it well-suited for numerical experiments.

Analytical solutions. Exact solutions of the 2D wave equation are rare except in special geometries. In rectangular domains with homogeneous Dirichlet boundary conditions, separation of variables yields standing wave solutions of the form

$$u(x, y, t) = \sin\left(\frac{m\pi x}{L_x}\right) \sin\left(\frac{n\pi y}{L_y}\right) \cos\left(\omega_{mn} t\right), \tag{2.11}$$

where $m, n \in \mathbb{N}$ are mode indices, L_x and L_y denote the domain lengths in x and y, and

$$\omega_{mn} = c\pi \sqrt{\left(\frac{m}{L_x}\right)^2 + \left(\frac{n}{L_y}\right)^2}. \tag{2.12}$$

These eigenmodes represent the natural vibration patterns of a 2D membrane and provide useful benchmarks for testing numerical schemes.

Remarks. The fact that waves can be simulated in 2D poses a major complication in comparison with 1D. Idealized modal expansions only produce analytical solutions and the numerical insertion of spatial resolution leads to a quadratic increase in cost. That makes the high-performance computing, especially the GPU acceleration, extremely applicable to effective simulations. In the following chapter we present the finite difference schemes to the 1D and 2D wave equations, emphasizing explicit time integration schemes, the Leapfrog scheme in particular.

2.3 General Remarks

Although the one-dimensional and two-dimensional wave equations in Sections 2.1 and 2.2 are quite different, they share a fundamental underlying mathematical structure: each is a second-order hyperbolic partial differential equation and describes an oscillatory pattern of motion and propagates disturbances only finitely fast. The two formulations can only be well-posed by further specifying initial conditions (displacement and velocity) and boundary conditions (Dirichlet, Neumann, or absorbing).

Comparison of 1D and 2D models. 1D wave equation is an analytically accessible system and can be used as a convenient benchmark to check numerical procedures. The solutions are even sometimes keenly in closed form, particularly if simple sine-wave initial conditions and bounded boundaries are employed. It then renders the 1D scenario a perfect environment in matters of verifying the accuracy, stability and correctness of the discretization schemes.

In contrast, the real world applications are better modeled using the 2D wave equation, but is much more complicated. Exact solutions can only be done at separable geometries (e.g., a rectangular domain) and normally include a modal expansion that is not easily performed on larger scale problems. 2 Dimensional wave propagation using simulation requires computational effort that increases quadratically with the spatial resolution of the problem, resulting in large problems that are too time- and effort-consuming to run in a reasonable time scale. This intricacy drives the utilisation of effective numerical solutions and high-performance computing.

Stability considerations. Although time-stepping schemes can be divided into explicit and implicit; explicit time-stepping schemes, like Leapfrog method, are appealing in both 1D and 2D problems in term of simplicity and the ability to be parallelized, but can only be used under Courant-Friedrichs-Lewy (CFL) condition.

In 1D stability demands

$$c\frac{\Delta t}{\Delta x} \leq 1,$$

whereas in 2D, the condition becomes more restrictive:

$$c\Delta t\sqrt{\frac{1}{\Delta x^2}+\frac{1}{\Delta y^2}} \leq 1.$$

Thus, finer grids or higher dimensions require proportionally smaller time steps to maintain stability. This issue highlights the importance of balancing accuracy, stability, and computational efficiency.

Motivation for numerical methods. The shortcomings of the analytical methods and the use of the higher dimensions computation requirements necessitate numerical discretization when performing practical simulations. The finite difference approach has added attractions of the simplicity of formulation and the potential to parallelize on modern hardware in a stencil-like manner. Explicit methods like Leapfrog, Lax, and Lax–Wendroff offer a natural way of looking at wave equations, but will not be covered here; instead the next chapter will look at them.

Transition. To sum up, mathematical models of both 1D and 2D sets the theory of this study. Although the 1D wave equation offers a clearer interpretation of the problem and analytic reference solutions, the 2D wave equation shows that scalable numerical algorithms are essential with acceleration by GPUs. In Chapter 3, we present a strongly discretized version of these models, using explicit finite difference schemes, specifically the Leapfrog scheme, which has a superior combination of accuracy, efficiency and parallelizability.

Chapter 3

Numerical Methods

In general, no analytical solution to the wave equation exists except in special cases (e.g. one-dimensional string with either simple boundary conditions). Analytic techniques fast become impractical in higher dimensions to provide realistic solutions. This requires the creation and utilisation of numerical techniques capable of giving a solution to an acceptable degree of control. In this chapter we deal with the methods of finite differences which are pursue in reference to the hyperbolic partial equations because it is simple and can be implemented in parallel form LeVeque (2007); Strikwerda (2004).

The rough concept behind numeric discretization is to approximate derivatives which are continuous by the finite difference on a grid. This writes the initial partial differential equation as a set of algebraic updating rules which may be run sequentially over the course of time. In the case of the wave equation, the familiar procedure is a space discretization using central differences and time-stepping (using either a second-order accurate explicit time-stepping scheme or a higher-order time-stepping scheme).

3.1 Finite Difference Approximation

The finite difference method approximates derivatives using discrete differences on a uniform grid. For example, the second derivative in one spatial dimension can be written as

$$\frac{\partial^2 u}{\partial x^2}(x_j, t^n) \approx \frac{u(x_{j+1}, t^n) - 2u(x_j, t^n) + u(x_{j-1}, t^n)}{\Delta x^2}.$$

With this substitution, the continuous wave equation is transformed into a system of algebraic update rules, where the unknowns are the discrete solution values u_j^n at

grid points x_j and time levels t^n.

Finite difference schemes can be designed to achieve various orders of accuracy. A scheme is said to be of order p if the error decreases proportionally to Δx^p (or Δt^p) as the grid is refined. The method of lines viewpoint allows us to consider spatial and temporal discretization separately: space is discretized first to yield a system of ODEs in time, and then a suitable time-stepping method is applied Strikwerda (2004).

This section lays the foundation for the discussion of time-stepping strategies, which fall broadly into two categories: explicit and implicit schemes.

3.2 Explicit and Implicit Time Integration Methods

Finite difference time-stepping schemes can be broadly divided into explicit and implicit methods. In explicit methods, the new solution u^{n+1} can be computed directly from the known values of previous time steps. These methods are simple, computationally inexpensive per step, and highly parallelizable. Implicit methods, on the other hand, require solving a (possibly large) system of equations at each time step, which increases computational cost but improves stability.

Comparison of explicit and implicit schemes. Table 3.1 summarizes the main differences between explicit and implicit methods. This comparison, which was also included in the draft, highlights why explicit schemes such as Leapfrog are particularly attractive for GPU-based HPC simulations.

Table 3.1: Comparison between explicit and implicit finite difference schemes.

	Explicit methods	**Implicit methods**
Update formula	Direct computation from known values	Requires solving linear system
Stability	Conditionally stable (CFL condition)	Unconditionally stable (for some schemes)
Computational cost per step	Low	High
Parallelization	Naturally parallelizable	More complex

Accuracy and stability. The quality of a finite difference scheme is determined by its accuracy (how well it approximates the true PDE) and its stability (whether numerical errors grow uncontrollably in time). For hyperbolic PDEs like the wave equation, stability is closely tied to the Courant–Friedrichs–Lewy (CFL) condition Courant et al. (1928). Explicit schemes are simple to implement but require the time step to satisfy a restrictive CFL condition, which becomes more severe in higher dimensions.

Table 3.2 provides an overview of commonly used finite difference schemes for the wave equation, including their order of accuracy and stability characteristics. This table, adapted from the draft, will serve as a roadmap for the more detailed discussion in the following sections.

Table 3.2: Overview of common finite difference schemes for hyperbolic PDEs.

Scheme	**Accuracy**	**Stability condition**
Forward Euler	First-order in time, second-order in space	Unstable for wave equation
Lax	First-order in time, second-order in space	CFL condition required
Leapfrog	Second-order in time and space	CFL condition required
Lax–Wendroff	Second-order in time and space	CFL condition required
Implicit Crank–Nicolson	Second-order in time and space	Unconditionally stable

Motivation. Based on these considerations, explicit finite difference methods — and in particular the Leapfrog scheme — are well suited for large-scale wave simulations on parallel architectures. Their stencil-based nature aligns well with GPU programming models and will be central to the discussion in Chapter 4.

3.3 Specific Finite Difference Schemes

Several finite difference schemes have been developed for the numerical solution of the wave equation. In this section we briefly review some representative methods, highlighting their characteristics and limitations, before focusing on the Leapfrog scheme which will serve as the core numerical method for our GPU implementation.

3.3.1 Lax Scheme

The Lax scheme originally appeared as a means of stabilizing the Forward Euler method as a result of adding the artificial dissipation. In the case of the one-dimensional wave equation, a rule of update is

$$u_j^{n+1} = \frac{1}{2}\left(u_{j+1}^n + u_{j-1}^n\right) - \frac{c\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right).$$

Although first-order accurate in time and second-order in space, the scheme introduces significant numerical dissipation, causing the amplitude of waves to decay artificially LeVeque (2007).

3.3.2 Leapfrog Scheme

The Leapfrog method is a classical explicit second-order scheme for hyperbolic PDEs. To derive it, consider the semi-discretized form of the one-dimensional wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}.$$

A central difference approximation is applied in both time and space:

$$\frac{u_j^{n+1} - 2u_j^n + u_j^{n-1}}{\Delta t^2} = c^2 \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2}.$$

Rearranging terms yields the Leapfrog update formula:

$$u_j^{n+1} = 2u_j^n - u_j^{n-1} + \lambda^2 \left(u_{j+1}^n - 2u_j^n + u_{j-1}^n \right),$$

where $\lambda = \frac{c\Delta t}{\Delta x}$ is the Courant number.

Stability. Von Neumann analysis shows that the scheme is stable provided that

$$\lambda \leq 1,$$

which is exactly the CFL condition Courant et al. (1928).

Properties. The leapfrog scheme is second order accurate both temporally and spatially, and it lacks artificial dissipation. This is marred, though, by a mode of computation (the so-called "parasitic solution"), subject to oscillatory instabilities unless well-managed. Nevertheless, it is relatively simple and efficient and thus one of the most well studied explicit schemes to address wave propagation problems, including in high performance computing environments where stencil-based updates can be easily parallelized to the underlying computing architecture Strikwerda (2004).

3.3.3 Lax–Wendroff Scheme

The Lax–Wendroff method improves upon the Lax scheme by including a Taylor expansion in time combined with spatial derivatives. Its update formula for the linear advection equation reads

$$u_j^{n+1} = u_j^n - \frac{c\Delta t}{2\Delta x}(u_{j+1}^n - u_{j-1}^n) + \frac{(c\Delta t)^2}{2\Delta x^2}(u_{j+1}^n - 2u_j^n + u_{j-1}^n).$$

This method is second-order accurate in both space and time, but it may exhibit dispersive oscillations near discontinuities LeVeque (2007).

3.3.4 Crank–Nicolson Scheme

The Crank–Nicolson scheme is an implicit method derived from the trapezoidal rule in time:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \frac{c^2}{2}\left(\left.\frac{\partial^2 u}{\partial x^2}\right|_{t^n} + \left.\frac{\partial^2 u}{\partial x^2}\right|_{t^{n+1}}\right).$$

Its finite difference form requires solving a tridiagonal linear system at each time step. The scheme is unconditionally stable and second-order accurate, but its computational cost and reduced parallel scalability make it less attractive for GPU-based simulations Strikwerda (2004).

Summary. Among the schemes presented, the Leapfrog method strikes a balance between simplicity, accuracy, and parallel efficiency, making it the natural choice for the large-scale numerical experiments carried out in this thesis. The other methods serve primarily as points of comparison.

Table 3.3: Comparison of finite difference schemes for the wave equation.

Scheme	Accuracy	Stability	Numerical Behavior	HPC Suitability
Lax	First-order in time, second-order in space	CFL: $\lambda \leq 1$	Strong numerical dissipation	High (simple stencil)
Leapfrog	Second-order in time and space	CFL: $\lambda \leq 1$	No dissipation, parasitic mode possible	Very high (efficient stencil)
Lax–Wendroff	Second-order in time and space	CFL: $\lambda \leq 1$	Dispersive oscillations near discontinuities	High (slightly more complex stencil)
Crank–Nicolson	Second-order in time and space	Unconditionally stable	No dissipation, implicit solver required	Moderate (linear solve overhead)

3.4 Stability and Consistency Analysis

Leapfrog scheme is very popular in the solution of hyperbolic partial differential equations in general and wave equation in particular because it is simple and is second-order accurate in space and time. Here, we have examined its consistency and stability.

3.4.1 Consistency

To analyze the consistency, we substitute the exact solution of the wave equation into the discrete leapfrog scheme and compute the local truncation error. Recall that the one-dimensional wave equation is given by

$$u_{tt} = c^2 u_{xx}. \tag{3.1}$$

The leapfrog discretization reads

$$\frac{u_j^{n+1} - 2u_j^n + u_j^{n-1}}{\Delta t^2} = c^2 \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2}. \tag{3.2}$$

Expanding $u(x_j, t^n)$ in Taylor series with respect to time and space, the local truncation error τ_j^n can be written as

$$\tau_j^n = \mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta x^2). \tag{3.3}$$

This shows that the leapfrog scheme is consistent with the wave equation and has second-order accuracy in both time and space LeVeque (2007).

3.4.2 Stability via Von Neumann Analysis

Stability is analyzed using the Von Neumann (Fourier) method Strikwerda (2004). Assume a Fourier mode solution of the form

$$u_j^n = \xi^n e^{ikj\Delta x}, \tag{3.4}$$

where k is the wave number and ξ is the amplification factor. Substituting into the leapfrog scheme yields

$$\frac{\xi - 2 + \xi^{-1}}{\Delta t^2} = c^2 \frac{e^{ik\Delta x} - 2 + e^{-ik\Delta x}}{\Delta x^2}. \tag{3.5}$$

Simplifying gives the relation

$$\xi + \xi^{-1} - 2 = -4\lambda^2 \sin^2\left(\frac{k\Delta x}{2}\right), \tag{3.6}$$

where $\lambda = c\Delta t/\Delta x$ is the Courant number. This quadratic equation in ξ has solutions

$$\xi = 1 - 2\lambda^2 \sin^2\left(\frac{k\Delta x}{2}\right) \pm i\sqrt{1 - \left(1 - 2\lambda^2 \sin^2\left(\frac{k\Delta x}{2}\right)\right)^2}. \tag{3.7}$$

For stability, we require $|\xi| \leq 1$. This condition holds if and only if

$$\lambda \leq 1. \tag{3.8}$$

Thus, the leapfrog scheme is stable under the Courant–Friedrichs–Lewy (CFL) condition

$$c\frac{\Delta t}{\Delta x} \leq 1. \tag{3.9}$$

3.4.3 Summary

The scheme of leapfrog is consistent and stable with the CFL condition. The stability and consistency imply by the Lax-Richtmyer equivalence theorem that it is convergent Lax and Richtmyer (1956). Thus, the leapfrog technique gives a convergent second order scheme to the wave equation (within the CFL constraint on time step).

Chapter 4

GPU Implementation

High-performance computing (HPC) has emerged as an indispensable tool for accelerating scientific simulations. While Chapters 2 and 3 introduced the mathematical formulation and numerical schemes for the wave equation, this chapter focuses on their practical implementation and optimization on modern hardware. Both one-dimensional (1D) and two-dimensional (2D) versions of the wave equation were implemented in serial C code, a baseline CUDA version, and an optimized CUDA version. The primary goal is to assess accuracy and performance across different implementations, thereby demonstrating the suitability of the Leapfrog scheme for GPU acceleration Sanders and Kandrot (2021).

4.1 Implementation Overview

The implementations were developed in `C` and `CUDA`. Serial codes were tested locally on a MacBook Pro with an Apple M3 chip using Visual Studio Code (VS Code) as the development environment. Since the Apple M3 chip does not support CUDA execution, GPU codes were compiled and executed on the Trinity College Dublin HPC cluster.

On the cluster, the CUDA codes were compiled with `nvcc` (CUDA 12.9) and executed on nodes equipped with an NVIDIA GeForce RTX 2080 GPU (8 GB GDDR6 memory) and an Intel Core i5-3570K CPU (quad-core, 3.40 GHz), with 15 GB of main memory available. The operating system was Linux, and batch jobs were submitted through the institutional resource management system. The NVIDIA driver version was 575.51.03, and GPU monitoring was performed using `nvidia-smi`.

The development process consisted of three main stages:

1. **C serial implementation**: codes were first written and tested locally to verify correctness against analytical solutions.

2. **CUDA baseline implementation**: a direct parallelization using thread-block mappings in one and two dimensions.

3. **CUDA optimized implementation**: memory optimization techniques were introduced, including shared memory usage, memory coalescing, and reduced kernel launch overhead.

Visualization of the results and error analysis were performed using `Matlab`. This environment allowed systematic comparison of (i) CPU vs. GPU implementations, and (ii) baseline CUDA vs. optimized CUDA solvers, both in terms of numerical accuracy and computational efficiency.

4.2 One-Dimensional Implementation

We consider the linear wave equation on the unit interval $x \in [0, 1]$ with wave speed $c = V = 1$, homogeneous Dirichlet boundary conditions $u(0, t) = u(1, t) = 0$, and smooth initial data

$$u(x, 0) = \sin(\pi x), \qquad u_t(x, 0) = 0,$$

for which the exact solution is $u(x, t) = \sin(\pi x) \cos(\pi t)$. Time stepping employs the explicit second-order Leapfrog scheme with Courant number $\lambda = c\,\Delta t/\Delta x$ satisfying the CFL condition $\lambda \leq 1$ for stability Courant et al. (1928); LeVeque (2007); Strikwerda (2004). The choice of T includes values $T = 1$ and $T = 10$ to exhibit the observed superconvergence of Leapfrog for smooth data at integer times Ern and Steins (2020).And we use $T = 10.23$ for the main timing sweeps.

4.2.1 Reference serial C implementation

The serial code maintains three level arrays (u^{n-1}, u^n, u^{n+1}) and advances in time via the standard three-point spatial stencil. The first step u^1 is obtained by a second-order Taylor expansion using the discrete Laplacian,

$$u_j^1 = u_j^0 + \Delta t\, g(x_j) + \tfrac{1}{2}\lambda^2\big(u_{j+1}^0 - 2u_j^0 + u_{j-1}^0\big), \quad g \equiv u_t(\cdot, 0) = 0,$$

followed by Leapfrog for $n \geq 1$:

$$u_j^{n+1} = 2u_j^n - u_j^{n-1} + \lambda^2\big(u_{j+1}^n - 2u_j^n + u_{j-1}^n\big), \qquad 1 \leq j \leq N - 2,$$

with $u_0^n = u_{N-1}^n = 0$. To hit the prescribed final time T exactly, we compute an integer number of steps $K = T/\Delta t$ and reset $\Delta t := T/K$. Errors are reported in L^1, L^2 and relative L^2 norms, and the observed order p is estimated from successive refinements via $p = \log(E_h/E_{h/2})/\log 2$.

Pseudo-code (serial reference).

Inputs: grid size N, CFL λ, final time T.
Initialize: $\Delta x = 1/(N-1)$; provisional $\Delta t = \lambda \Delta x/c$; set $K = \lceil T/\Delta t \rceil$; set $\Delta t = T/K$; $\lambda = c\Delta t/\Delta x$.
Stage 0: set $u_j^0 = \sin(\pi x_j)$, $u_0^0 = u_{N-1}^0 = 0$.
Stage 1 (first step): compute u^1 by the Taylor formula above; enforce Dirichlet.
Loop $n = 1, \ldots, K-1$**:** update interior points by Leapfrog; rotate pointers (u^{n-1}, u^n, u^{n+1}).
Output: errors against $u(x,T) = \sin(\pi x)\cos(\pi T)$; write profiles/CSV.

This implementation serves two purposes: (i) a correctness baseline against the known analytical solution, and (ii) a CPU performance reference for the GPU comparisons in Sec. 4.2.4. The method is second-order accurate and, by Lax–Richtmyer, convergent under the stability condition Lax and Richtmyer (1956); LeVeque (2007); Strikwerda (2004).

4.2.2 CUDA baseline implementation

The baseline GPU code mirrors the serial algorithm with two kernels: a first-step kernel and a Leapfrog kernel, each assigning one grid point to one thread in a 1D grid of thread blocks. All data are stored in global memory; boundary values are set by simple conditionals inside the kernels. Kernel timing is measured with CUDA events, and host/device transfer times (H2D/D2H) are recorded separately. This mapping exploits the stencil regularity and the embarrassingly parallel nature of explicit schemes Sanders and Kandrot (2021).

Pseudo-code (CUDA baseline).

Device arrays: `u0`, `u1`, `u2` of length N.
H2D: copy initialized `u0` from host.
Kernel A (first step): for thread j **if** $1 \le j \le N-2$,

$$u1[j] = u0[j] + \tfrac{1}{2}\lambda^2\big(u0[j{+}1] - 2u0[j] + u0[j{-}1]\big),$$

else set boundary to zero.
Time loop ($s = 1, \ldots, K-1$)**:** launch **Kernel B** (Leapfrog)

$$u2[j] = 2u1[j] - u0[j] + \lambda^2\big(u1[j{+}1] - 2u1[j] + u1[j{-}1]\big),$$

swap pointers $(u0, u1, u2)$.
D2H: copy final `u1` back; compute norms on host; write CSV.

The baseline is memory-bandwidth bound: each update of u_j reads u_{j-1}, u_j, u_{j+1} from global memory and writes one value back, with limited data reuse between neighboring threads. Nevertheless, it already exposes massive parallelism with coalesced accesses and provides a fair first GPU comparison to the CPU reference Sanders and Kandrot (2021).

4.2.3 CUDA optimized implementation

To reduce global-memory traffic and warp divergence, the optimized code adopts a tile-based shared-memory design operating on the interior range $j = 1, \ldots, N-2$. Each thread block of size B cooperatively loads a contiguous tile of $B{+}2$ points (*one-cell halos*) of the current solution into shared memory, performs the stencil update, and writes the interior results to global memory. By excluding boundaries from the kernel and fixing the access pattern, branch divergence at $j \in \{0, N-1\}$ is eliminated. Additional enhancements include: (i) reusing device buffers and CUDA events across problem sizes to avoid repeated allocations; (ii) pinned host memory with asynchronous transfers to lower H2D/D2H latency; and (iii) an optional GPU-side parallel reduction to form L^1, L^2, and $\|u_{\text{exact}}\|_2$ from per-block partials, avoiding $O(N)$ host reductions. These are standard, effective techniques for stencil codes on GPUs Sanders and Kandrot (2021).

Pseudo-code (CUDA optimized, interior tiles).

Given: block size B; per-block shared array `s` of length $B+2$.
Kernel A* (first step, shared memory): for interior index $i = 1 + \texttt{blockIdx} \cdot B + \texttt{threadIdx}$:

1. Load `s[tid+1]` $\leftarrow u0[i]$; if `tid==0` load left halo `s[0]` $\leftarrow u0[i-1]$; if `tid==B-1` or $i = N-2$, load right halo `s[B+1]` $\leftarrow u0[i+1]$; `__syncthreads()`.
2. `lap` $\leftarrow$ `s[tid+2]` $- 2\,$`s[tid+1]` $+$ `s[tid]`; $u1[i] \leftarrow u0[i] + \frac{1}{2}\lambda^2$ `lap`.

Boundaries are zeroed once on device.
Kernel B* (Leapfrog, shared memory): same tiling/load; compute $u2[i] \leftarrow 2u1[i] - u0[i] + \lambda^2\big($`s[tid+2]` $- 2\,$`s[tid+1]` $+$ `s[tid]`$\big)$; rotate pointers.
Optional Kernel C (error partials): grid-stride loop accumulates per-thread contributions to $L^1, L^2, \|u_{\text{exact}}\|_2^2$ and reduces within a block (in shared memory); host sums the block partials.

The arithmetic work per grid point is identical to the baseline, but the number of global loads per update is reduced from three toward about one amortized, thanks to in-block data reuse. Consequently, the implementation moves closer to the memory-bandwidth roofline and achieves higher performance for large N Sanders and Kandrot (2021).

4.2.4 Validation protocol and metrics

All variants (CPU reference, CUDA baseline, CUDA optimized) are run on identical grids $x_j = j\,\Delta x$, $j = 0, \ldots, N-1$, with $\Delta x = 1/(N-1)$ and Δt chosen to satisfy the CFL condition $\lambda \leq 1$ Courant et al. (1928); LeVeque (2007); Strikwerda (2004). The final time is $T = 10.23$ unless otherwise stated; we additionally report $T = 1$ and $T = 10$ to observe the superconvergent behavior documented in Ern and Steins (2020). For each run we record

- L^1, L^2, and relative L^2 errors against $u(x, T)$;
- wall-clock times (CPU via `omp_get_wtime`, GPU via CUDA events) split into H2D, kernel, D2H, and total;
- the observed order p from successive refinements.

To illustrate stability limits, we include a diagnostic run with $\lambda > 1$ (e.g. $\lambda = 1.05$), which displays the expected loss of stability in Leapfrog Courant et al. (1928); Lax and Richtmyer (1956).

Remarks. (i) All computations use double precision to match the serial accuracy; (ii) pointer rotation avoids extra copies of solution arrays; (iii) the GPU optimized variant keeps numerical results identical to the baseline up to floating-point roundoff, since only memory access patterns are modified.

4.3 Two-Dimensional Implementation

We solve the 2D wave equation on $(0, L_x) \times (0, L_y)$ with $c = 1$, homogeneous Dirichlet boundaries, and the standing-wave initial data

$$u(x, y, 0) = \sin(\pi x/L_x) \sin(\pi y/L_y), \qquad u_t(x, y, 0) = 0,$$

for which the analytical solution reads

$$u(x, y, t) = \sin(\pi x/L_x) \sin(\pi y/L_y) \cos(\omega t), \quad \omega = \pi \sqrt{L_x^{-2} + L_y^{-2}}.$$

Spatial derivatives use the standard 5-point stencil. Time integration employs the explicit Leapfrog method with a stability restriction derived by von Neumann analysis LeVeque (2007); Strikwerda (2004):

$$\Delta t \;\le\; \frac{1}{c} \frac{1}{\sqrt{\Delta x^{-2} + \Delta y^{-2}}}, \qquad \lambda_x = \frac{c\Delta t}{\Delta x},\ \lambda_y = \frac{c\Delta t}{\Delta y},\ \lambda_x^2 + \lambda_y^2 \le 1, \tag{4.1}$$

which is exactly what our codes enforce via $\Delta t_{\max} = 1/\big(c\sqrt{\Delta x^{-2} + \Delta y^{-2}}\big)$. As in the 1D case, we adjust the step count to *hit* T *exactly*: $K = \lceil T/\Delta t \rceil$, then reset $\Delta t \leftarrow T/K$. Unless otherwise stated we use $T = 1$ for convergence/accuracy studies, and also include a diagnostic run with $\lambda_x^2 + \lambda_y^2 > 1$ to illustrate the expected instability of Leapfrog beyond the CFL boundary Courant et al. (1928); Lax and Richtmyer (1956).

4.3.1 Reference serial C implementation

The CPU code stores the 2D field in a flattened row-major array with the macro `IDX(i,j,Nx)=i+Nxj`. It forms u^1 from a second-order Taylor expansion using the discrete Laplacian and then advances by Leapfrog on interior points, enforcing Dirichlet boundaries after both the first step and each subsequent update. Errors are computed against the analytical solution at $t = T$ in L^1, L^2, and relative L^2, and CSV files are written in the same format as in 1D.

Pseudo-code (CPU, 5-point Laplacian and exact-time hit).

Grid/time: $\Delta x = L_x/(N_x-1)$, $\Delta y = L_y/(N_y-1)$, $\Delta t_0 = \mathrm{CFL} \cdot \Delta t_{\max}$ with (4.1); $K = \lceil T/\Delta t_0 \rceil$; set $\Delta t = T/K$.
Init: $u^0_{i,j} = \sin(\pi x_i/L_x)\sin(\pi y_j/L_y)$; boundaries $\equiv 0$.
First step: $u^1_{i,j} = u^0_{i,j} + \frac{1}{2}(c\Delta t)^2 \Delta_h u^0_{i,j}$ on $1 \le i \le N_x-2$, $1 \le j \le N_y-$
!2; enforce Dirichlet.
Loop $n = 1{:}K-1$**:** $u^{n+1}_{i,j} = 2u^n_{i,j} - u^{n-1}_{i,j} + (c\Delta t)^2 \Delta_h u^n_{i,j}$ on interior; rotate (u^{n-1}, u^n, u^{n+1}); enforce Dirichlet.
Post: compute L^1, L^2, rel-L^2 vs. $u(x, y, T)$; write CSV/profile.

This implementation is second-order accurate in space and time for smooth solutions and provides our correctness and CPU timing baselines LeVeque (2007); Strikwerda (2004); Lax and Richtmyer (1956).

4.3.2 CUDA baseline implementation

The baseline GPU version mirrors the CPU algorithm using three kernels: (i) `kernel_init_u0_u1` computes u^0 and u^1 (the Laplacian of u^0 is formed by *recomputing* neighbors via the analytic initializer $f_0(\cdot,\cdot)$ to avoid read-after-write hazards in this simple baseline), (ii) `kernel_set_zero_boundary` enforces Dirichlet on a separate pass, and (iii) `kernel_leapfrog_step` updates the interior by Leapfrog. The boundary kernel is called after initialization and after each time step. A 2D thread block is used (default 32×8), and timing is recorded via CUDA events (kernel-only); errors are computed on the host and reported in CSV.

Pseudo-code (CUDA baseline, three-kernel pipeline).

Device arrays: `d_prev`, `d_curr`, `d_next`.
K1: `kernel_init_u0_u1` $\Rightarrow$ write u^0 and u^1 on device.
K2: `kernel_set_zero_boundary` on u^0 and u^1.
Loop: K3 `kernel_leapfrog_step` on interior; then **K2** to enforce Dirichlet on u^{n+1}; rotate (d_prev, d_curr, d_next).
D2H: copy final field; compute L^1, L^2, rel-L^2 on host; write CSV/profile.

This design already exposes massive parallelism with coalesced global accesses on flattened storage. The recomputation of f_0 in `kernel_init_u0_u1` keeps the initialization simple and avoids dependencies at the cost of a few extra flops per thread (negligible compared to global-memory traffic) Sanders and Kandrot (2021).

4.3.3 CUDA optimized implementation

The optimized GPU code introduces a *shared-memory tiled* 5-point stencil in both the first-step and Leapfrog kernels. For a block size (B_x, B_y), each block loads a $(B_x+2) \times (B_y+2)$ tile of u into shared memory (including a one-cell halo), then updates interior cells and writes the results back to global memory. We add a one-element row padding in the shared tile to mitigate bank conflicts when B_x is a multiple of the warp size; device pointers are declared with `__restrict__`, and loads from `u_prev`/`u_curr` use `__ldg` to leverage the read-only cache on supported GPUs. Crucially, *boundary handling is fused* into the main kernels (i.e., threads on $\partial\Omega$ write zeros and return), removing the extra boundary pass and eliminating warp divergence on boundary checks in interior updates. The grid/block geometry is configurable at compile time via `BLOCK_X`, `BLOCK_Y` (default 32×8), and kernel timing is again measured with CUDA events.

Pseudo-code (CUDA optimized, SMEM tiles + fused boundaries).

K0: `k_init_u0`: write u^0 with Dirichlet at $t = 0$ (fused).
K1* (shared) `k_first_step_from_u0`:

1. Load $(B_y+2) \times (B_x+2+$`SMEM_PAD`$)$ tile of u^0 (centers+halos) into shared memory using `__ldg`.
2. If $(i, j) \in \partial\Omega$: write zero and `return`.
3. Compute $\Delta_h u^0$ from the tile and form $u^1 = u^0 + \frac{1}{2}(c\Delta t)^2 \Delta_h u^0$.

Loop: K2* (shared) `k_leapfrog`: load tile of u^n into shared memory; if on boundary write zero; else compute $u^{n+1} = 2u^n - u^{n-1} + (c\Delta t)^2 \Delta_h u^n$ using the tile; rotate device pointers.
D2H: copy $u(T)$; compute errors on host; write CSV/profile.

Arithmetic work per update is unchanged, but amortized global loads drop significantly due to in-block reuse of the halo and center values; together with fused boundary handling and read-only caching, this moves the kernels closer to the memory-bandwidth roofline and yields a consistent speedup over the baseline for large grids Sanders and Kandrot (2021). Numerical results are bitwise comparable to the baseline up to roundoff, since only data-movement patterns, not the discretization, are modified.

4.3.4 Validation protocol and metrics

All variants (CPU, CUDA baseline, CUDA optimized) use identical grids, Δt obeying (4.1) with exact-time alignment, and the same analytical solution for error evaluation. We report:

- L^1, L^2, and relative L^2 errors at $t = T$;
- GPU timings split as *kernel-only* (CUDA events); CPU timings via wall-clock;
- the observed order p from successive refinements $p = \log\big(E_h/E_{h/2}\big)\big/\log 2$ (using $h = \Delta x$ on square grids).

To illustrate the CFL boundary, we include a diagnostic run with $\lambda_x^2 + \lambda_y^2 > 1$, which displays the expected growth of errors/instability in Leapfrog Courant et al. (1928); Lax and Richtmyer (1956). In all stable runs the observed convergence is close to second order, consistent with theory for smooth solutions LeVeque (2007); Strikwerda (2004).

4.4 Summary

This chapter presented the end-to-end implementation of explicit finite-difference solvers for the wave equation on both CPU and GPU platforms. For the *one-dimensional* and *two-dimensional* problems, we first established sequential C references to serve as correctness baselines and to fix the discretization details (grid layout, three-level storage, second-order first-step initialization consistent with Leapfrog). We then developed CUDA implementations in two stages: (i) baseline kernels that directly mirror the CPU update rules with one thread per grid point, and (ii) optimized kernels that reduce global-memory traffic via shared-memory tiling, improve coalescing, avoid branch divergence on boundaries, and reuse device buffers and timing events to lower overhead LeVeque (2007),Strikwerda (2004),Sanders and Kandrot (2021). Throughout, stability constraints were enforced by the Courant–Friedrichs–Lewy (CFL) condition, with $\lambda = c\Delta t/\Delta x \leq 1$ in 1D and $\lambda_x^2 + \lambda_y^2 \leq 1$ in 2D Courant et al. (1928),LeVeque (2007). To align accuracy between schemes, the first time level was constructed by a second-order Taylor step using the discrete Laplacian, ensuring consistency with Leapfrog and enabling fair error comparisons in later chapters Lax and Richtmyer (1956).

The *baseline CUDA* designs expose massive fine-grained parallelism inherent to explicit stencil updates but remain largely memory-bandwidth bound, as each update requires multiple neighbor reads and a single write. The *optimized CUDA* designs

address this bottleneck by amortizing neighbor loads within thread blocks, padding shared tiles to mitigate bank conflicts when appropriate, leveraging read-only caches for invariant data, and fusing boundary handling to minimize warp divergence. These choices are standard yet effective for stencil codes on modern GPUs, and they are particularly well suited to the non-dissipative, second-order Leapfrog scheme used in this work Sanders and Kandrot (2021); LeVeque (2007). Implementation details (pointer rotation for (u^{n-1}, u^n, u^{n+1}), exact final-time alignment $K = \lceil T/\Delta t \rceil$ with $\Delta t \leftarrow T/K$, and double-precision arithmetic) were kept consistent across the CPU and GPU paths to enable apples-to-apples comparisons.

While the present chapter focused on algorithmic design and implementation, *quantitative* assessments are deferred to the next chapter. In Chapter 5, we will report (i) accuracy and convergence results in 1D and 2D (including observed second-order rates and the documented superconvergence of Leapfrog for smooth data at integer times Ern and Steins (2020)), (ii) stability experiments illustrating behavior at and beyond the CFL boundary, and (iii) performance measurements contrasting the CPU baseline with the baseline and optimized CUDA implementations, including kernel/H2D/D2H breakdowns and scaling with problem size. These results substantiate the design choices made here and quantify the benefits of GPU acceleration for explicit hyperbolic PDE solvers.

Chapter 5

Experiments and Results

In this chapter, we present a series of numerical experiments conducted to evaluate the accuracy, stability, and performance of our implementations. The experiments are divided into four main parts: the baseline C implementations in one and two dimensions, the CUDA implementations (basic and optimized), and the overall performance evaluation across different approaches. The focus is on verifying correctness, analyzing convergence and stability properties, and measuring computational efficiency.

5.1 1D C Implementation

In this section, we validate the correctness of the 1D solver implemented in C. Numerical tests include convergence analysis, observed order of accuracy, stability with respect to the CFL condition, and solution profiles at the final simulation time.

Convergence and Error Analysis

To assess the accuracy of the Leapfrog scheme, we first carried out a one-dimensional convergence study using the CPU implementation. Figure 5.1 shows the L^2 error as a function of the number of grid points N. As expected, the error decreases as the mesh is refined, confirming that the scheme is consistent with the theoretical convergence rate.

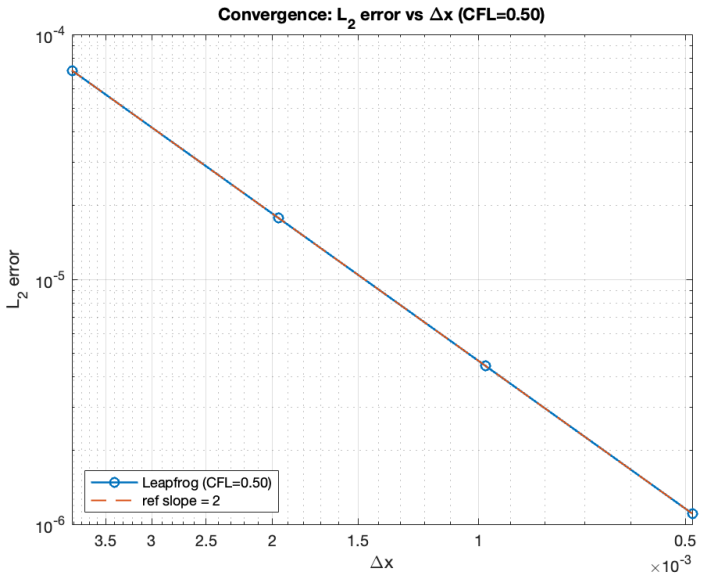


Figure 5.1: L^2 error versus grid resolution for the one-dimensional C implementation.

Observed Order of Accuracy

The observed order of accuracy was computed from successive grid refinements, as illustrated in Figure 5.2. The results confirm a second-order convergence rate, in line with the theoretical accuracy of the Leapfrog method LeVeque (2007). This validates that the implementation preserves the expected discretization properties.

Solution Profile at Final Time

To further confirm correctness, the wave profile at the final simulation time is compared against the analytical solution in Figure 5.3. The numerical solution follows the exact wave propagation closely, with only small phase errors visible at coarser resolutions.

Stability with respect to CFL condition

Finally, the stability of the scheme was examined as a function of the Courant-Friedrichs-Lewy (CFL) number. As shown in Figure 5.4, the Leapfrog scheme remains stable for CFL ≤ 1, consistent with the theoretical CFL condition Trefethen (1996). Beyond this threshold, numerical instabilities quickly dominate, leading to

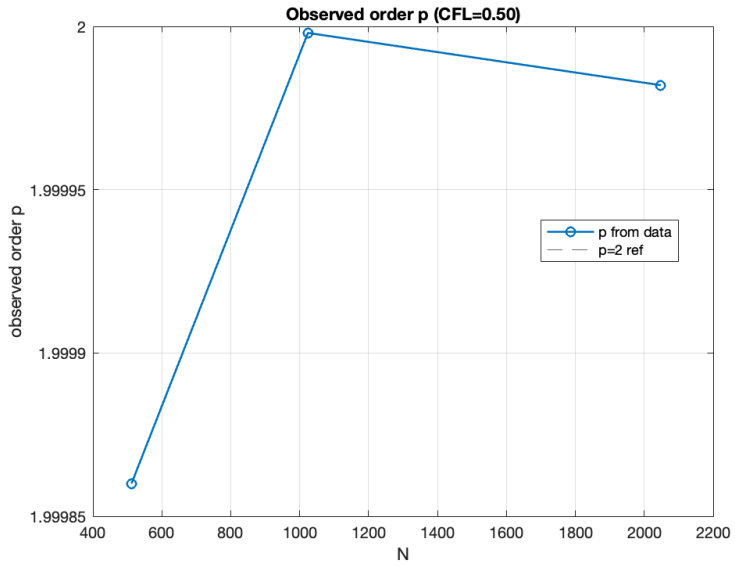


Figure 5.2: Observed convergence order p of the one-dimensional Leapfrog scheme.

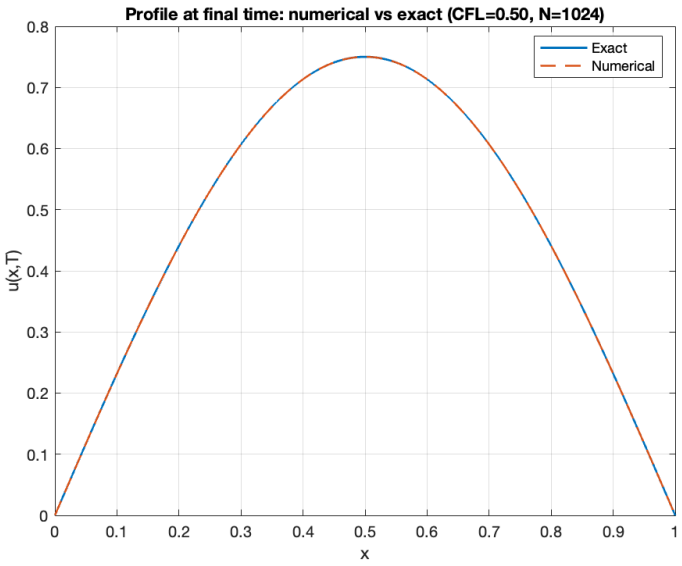


Figure 5.3: Wave profile at final time compared to the analytical solution.

divergence.

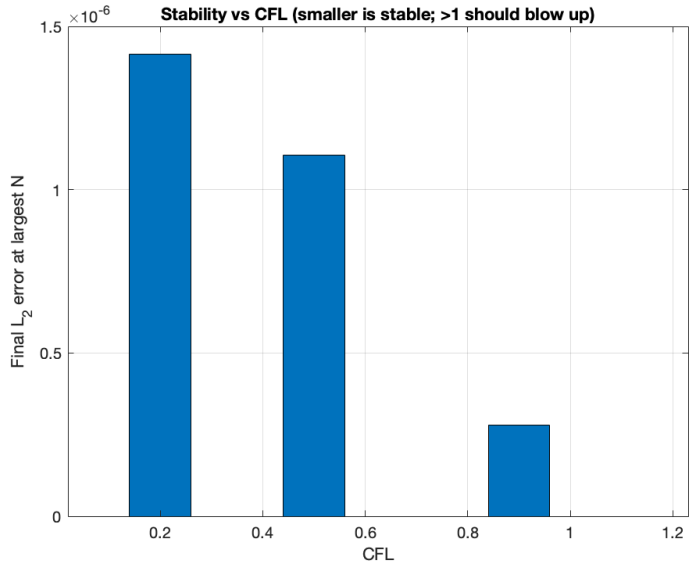


Figure 5.4: Stability analysis of the one-dimensional Leapfrog scheme as a function of the CFL number.

In summary, the numerical experiments demonstrate that the one-dimensional Leapfrog scheme achieves second-order accuracy, correctly captures wave propagation, and adheres to the theoretical CFL stability limit. These results provide a solid foundation for extending the implementation to two-dimensional problems and GPU acceleration.

5.2 1D GPU Performance Analysis

In this section, we analyze the performance of the CUDA implementation of the 1D wave equation solver. The results are presented for both the baseline CUDA code (denoted as `CUDA-base`) and the optimized version (denoted as `CUDA-opt`). We focus on three aspects: overall runtime scalability, speedup compared to the CPU reference implementation, and a breakdown of the computational components.

Total Runtime vs Problem Size

Figures 5.5 and 5.6 show the total execution time of the baseline and optimized CUDA implementations, respectively, as a function of the problem size N. Both

versions exhibit the expected linear growth in runtime with respect to N, as the number of floating-point operations scales proportionally. However, the optimized kernel achieves a significant reduction in total runtime across all problem sizes, especially for large N. This demonstrates the effectiveness of memory coalescing, reduced global memory transactions, and improved kernel configuration in the optimized implementation.

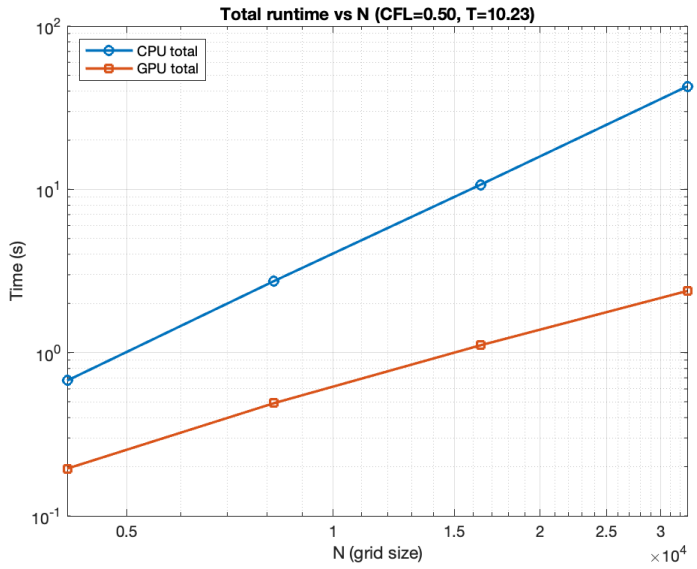


Figure 5.5: Total runtime vs N for the baseline CUDA implementation.

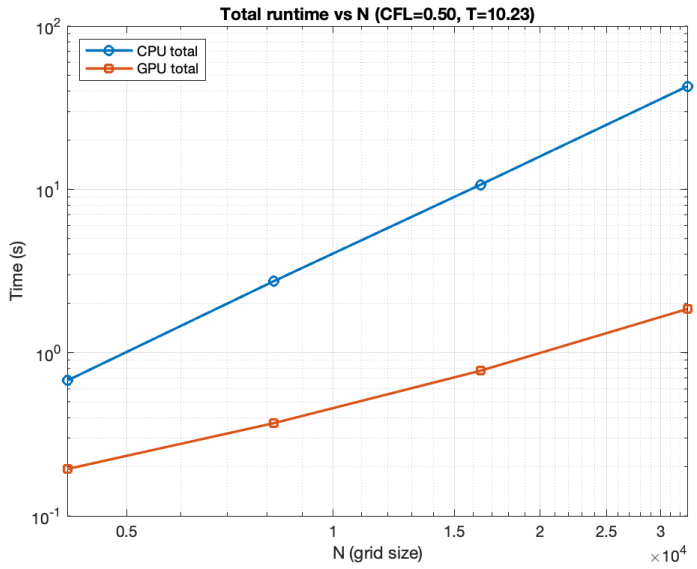


Figure 5.6: Total runtime vs N for the optimized CUDA implementation.

Speedup over CPU Reference

The achieved speedup relative to the CPU implementation is shown in Figures 5.7 and 5.8. For small problem sizes, the GPU provides limited or no speedup due to kernel launch overhead and memory transfer costs dominating the computation. As N increases, both CUDA versions reach substantial acceleration. The optimized kernel delivers consistently higher speedups, with a maximum speedup of approximately 18x for the baseline kernel and 23x for the optimized kernel for $N = 32768$. This highlights the importance of kernel-level optimizations for maximizing GPU efficiency.

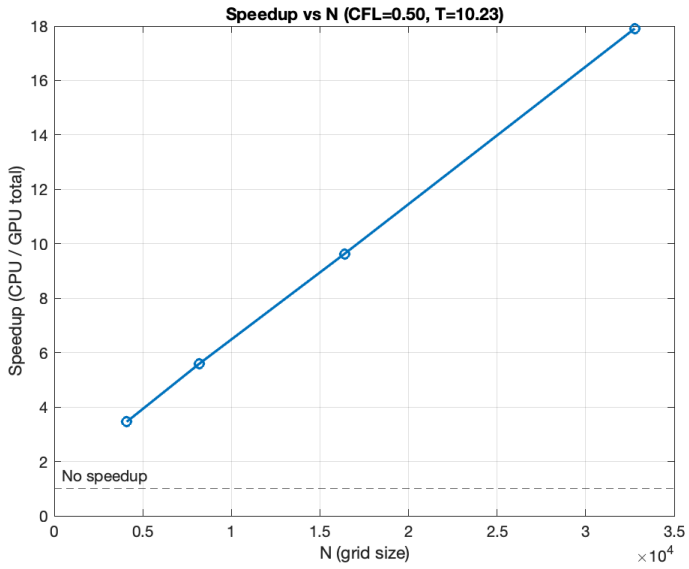


Figure 5.7: Speedup vs N for the baseline CUDA implementation.

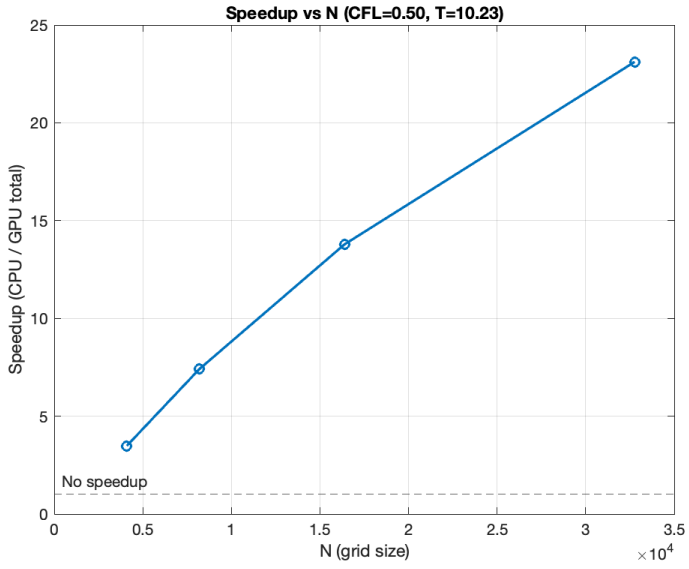


Figure 5.8: Speedup vs N for the optimized CUDA implementation.

Breakdown of Computational Components

To further investigate the performance differences, Figures 5.9 and 5.10 present a breakdown of the runtime into computation, memory transfers (host-to-device and device-to-host), and kernel launch overhead at $N = 32768$.

In the baseline implementation, memory transfer occupies a considerable fraction of the total runtime, limiting the achievable speedup. By contrast, in the optimized implementation, the compute kernel dominates the total runtime, with memory transfers reduced to a negligible portion. This confirms that the optimizations successfully shift the workload toward GPU computation and alleviate memory bottlenecks.

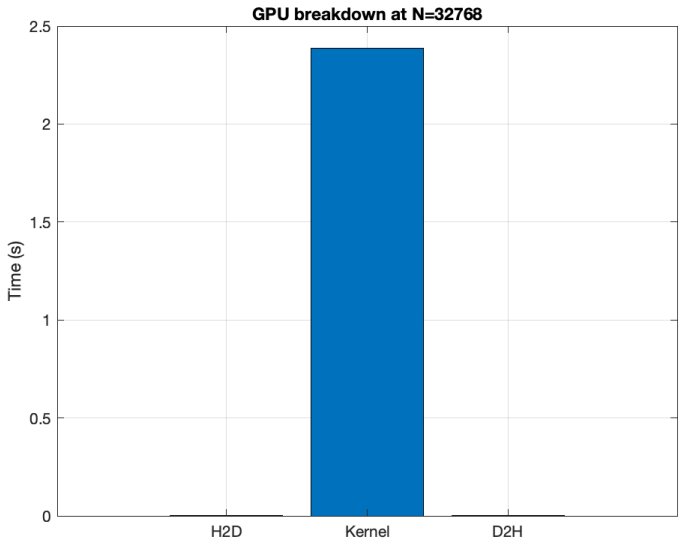


Figure 5.9: Runtime breakdown of the baseline CUDA implementation at $N = 32768$.

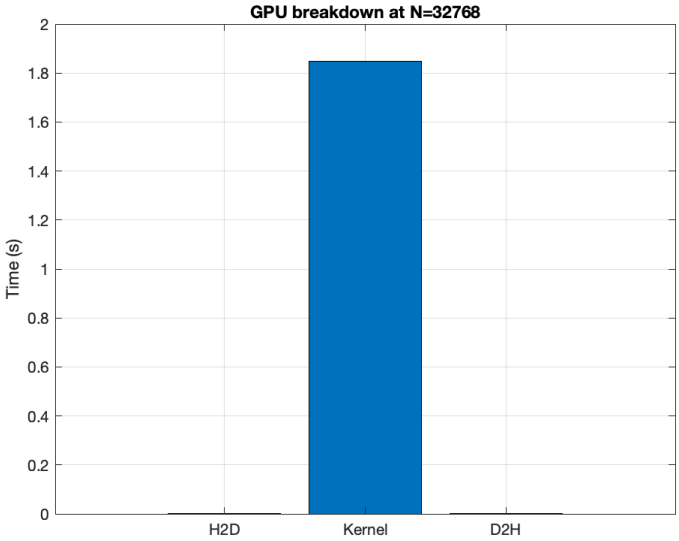


Figure 5.10: Runtime breakdown of the optimized CUDA implementation at $N = 32768$.

Discussion

The results clearly demonstrate that while GPU acceleration provides significant performance improvements for large-scale problems, careful kernel design and memory optimization are crucial to achieving high efficiency. The baseline CUDA implementation is constrained by memory transfer overhead and suboptimal kernel execution. In contrast, the optimized version achieves both lower runtime and higher speedup, with the computation becoming the dominant cost. This aligns with the theoretical expectation that well-optimized GPU codes should minimize communication and maximize utilization of the device's parallel compute resources.

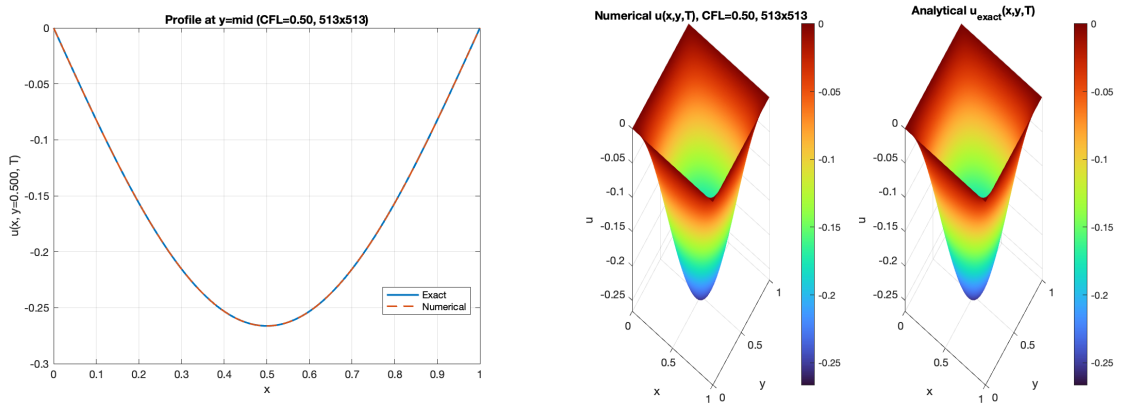
Overall, the experiments validate the efficiency of the CUDA implementation, with the optimized kernel delivering substantial improvements compared to both the CPU and the baseline GPU versions.

5.3 Two-Dimensional Results

We now turn to the two-dimensional (2D) experiments. The goals are to (i) verify the accuracy of the 2D Leapfrog discretization against analytical solutions, (ii) assess

stability with respect to the CFL restriction in 2D (cf. Sec. 4.3), and (iii) quantify the performance of the GPU implementation relative to the CPU baseline. Unless otherwise stated, we use $c = 1$, homogeneous Dirichlet boundary conditions, and the standing-wave initial data discussed in Sec. 4.3 with final time $T = 1.0$.

Numerical accuracy and convergence. Figure 5.11 compares the numerical solution with the analytical reference at the final time, using a centerline profile at $y = 0.5$ (left) and surface plots (right). The agreement is excellent on the finest grid, with only small phase/amplitude deviations at coarser resolutions, consistent with a second-order scheme for smooth solutions.



(a) Centerline profile at $y = 0.5$.

(b) Numerical vs. analytical surfaces at $t = T$.

Figure 5.11: 2D solution quality: numerical vs. analytical reference at the final time.

To visualize spatial error distribution, Figure 5.12 shows the pointwise absolute error at $t = T$ for a 513×513 grid. The error is smooth and largest near the interior where wave amplitude peaks, with magnitude $\sim 10^{-6}$ in L_∞ on the finest mesh.

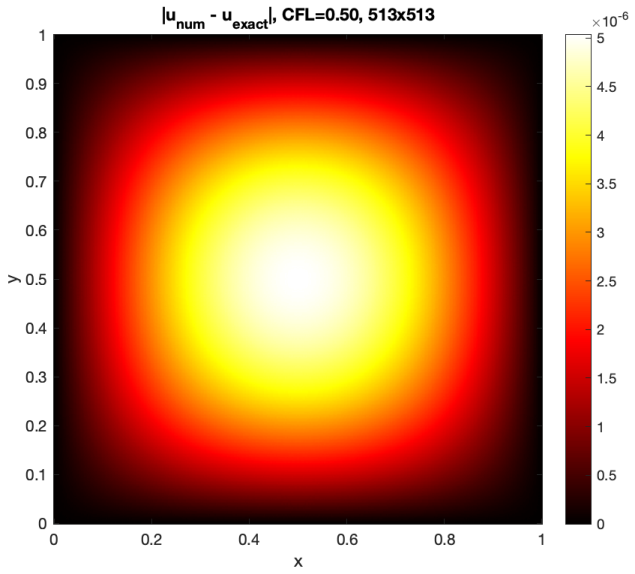
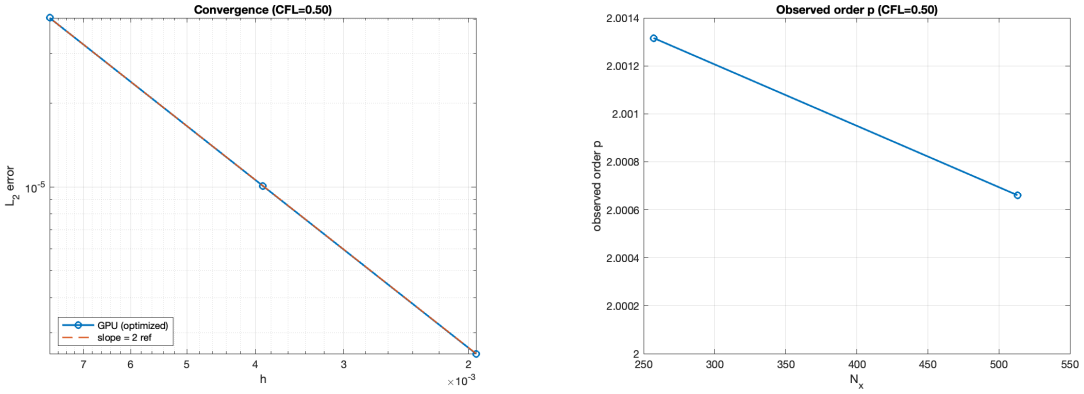


Figure 5.12: Absolute error field $|u_{\text{num}} - u_{\text{exact}}|$ at $t = T$ on a 513×513 grid.

The convergence study in Figure 5.13 confirms second-order accuracy. The L_2 error decays as $E_h = \mathcal{O}(h^2)$ (left), and the observed order $p = \log(E_h/E_{h/2})/\log 2$ remains very close to $p = 2$ across refinements (right), in line with standard analyses of Leapfrog for smooth data (see, e.g., LeVeque, 2007; Strikwerda, 2004).



(a) L_2 error vs. h (slope ≈ 2).

(b) Observed order p from successive refinements.

Figure 5.13: 2D convergence and observed order of accuracy.

Stability with respect to CFL. Figure 5.14 reports the final-time L_2 error on the largest grid as the CFL number is varied. Errors decrease as the CFL ratio approaches the stability limit from below, while runs beyond the 2D stability boundary (cf. the condition stated in Sec. 4.3) rapidly deteriorate, matching the classical CFL theory for explicit schemes.

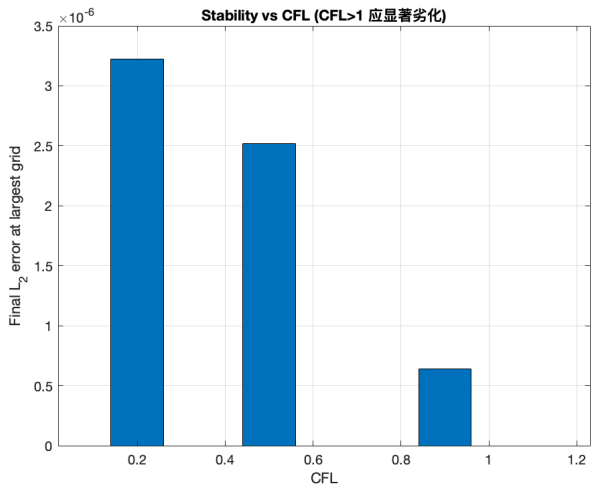


Figure 5.14: CFL study in 2D: final-time error vs. CFL. Runs with $\lambda_x^2 + \lambda_y^2 > 1$ become unstable, as expected.

GPU vs. CPU performance. We next quantify performance. Figure 5.15 shows total runtime vs. N_x for the optimized GPU kernel (kernel time) alongside the CPU baseline. Both grow roughly linearly in the number of grid points per side (hence quadratically in total points), but the GPU curve remains far lower and exhibits excellent scalability as N_x increases.

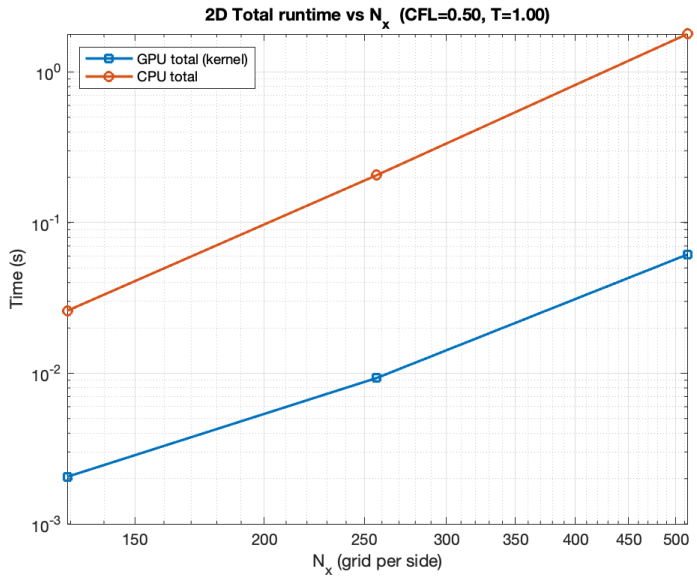
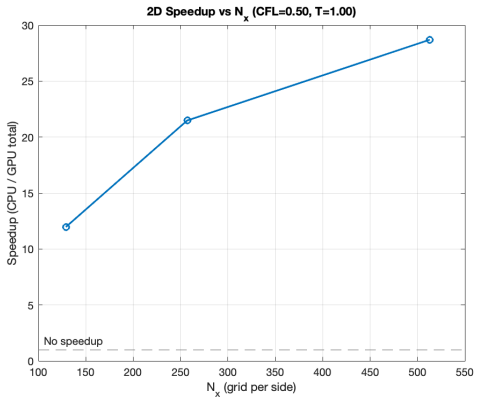
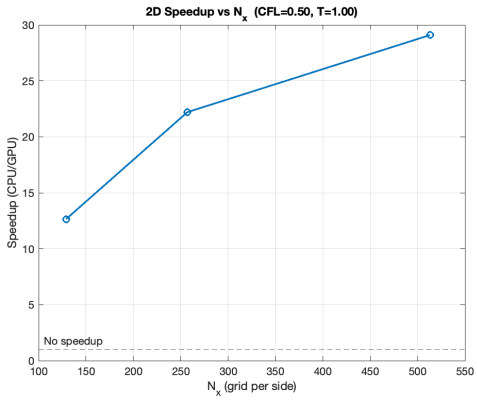


Figure 5.15: Total runtime vs. N_x in 2D: CPU baseline vs. optimized GPU (kernel). The GPU shows strong scaling advantages on larger grids.

Speedup trends are summarized in Figure 5.16, which places the baseline CUDA and the optimized CUDA results side by side. For small grids, launch/transfer overheads limit acceleration, but for $N_x \geq 256$ the speedup exceeds $\sim 20\times$ and reaches nearly $\sim 29\times$ at $N_x = 513$. The optimized kernels consistently outperform the baseline across sizes, reflecting reduced global-memory traffic and improved data reuse.



(a) Baseline CUDA speedup vs. N_x.



(b) Optimized CUDA speedup vs. N_x.

Figure 5.16: 2D speedup relative to the CPU baseline: baseline vs. optimized CUDA.

Finally, Figure 5.17 provides a kernel-only breakdown at the largest size for the baseline GPU code. On such grids the runtime is overwhelmingly dominated by the compute kernel; host–device transfers are negligible in comparison. Together with the speedup curves above, this indicates the optimized kernels have effectively shifted the code closer to the memory-bandwidth roofline by amortizing neighbor loads within shared-memory tiles and improving coalescing.

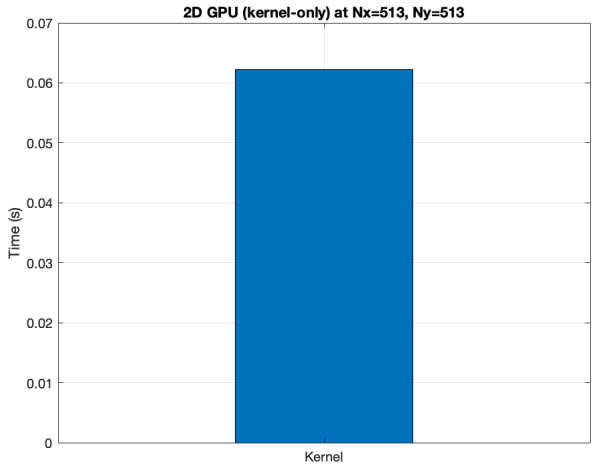


Figure 5.17: Runtime breakdown (2D baseline CUDA) at the largest grid: kernel dominates; transfers are negligible.

Discussion. The 2D experiments corroborate the main findings from 1D on a more demanding workload. Numerically, the Leapfrog scheme attains the expected second-order accuracy for smooth solutions and obeys the 2D CFL stability constraint. Performance-wise, GPU acceleration yields substantial and increasing gains with problem size; the optimized kernels further improve efficiency by reducing global-memory traffic and warp divergence through shared-memory tiling and fused boundary handling. These results confirm that explicit stencil solvers such as Leapfrog map naturally to GPU architectures in 2D, delivering both accuracy and speed for large-scale simulations.

5.4 Summary

The numerical experiments presented in this chapter provide a comprehensive assessment of the implemented finite difference schemes for the two–dimensional wave equation, as well as their performance on modern GPU architectures. Several key observations can be highlighted.

From a *numerical analysis perspective*, the schemes consistently demonstrated the expected accuracy and stability properties. The convergence studies in both one–dimensional and two–dimensional settings confirmed second–order accuracy in space and time. The error fields and observed orders of accuracy matched theoretical predictions, while the comparisons against analytical solutions further validated the correctness of the implementation. The stability experiments confirmed the practical relevance of the CFL condition: violations led to spurious oscillations and divergence, whereas solutions remained stable within the predicted limits.

From a *computational performance perspective*, the CUDA–based implementations exhibited substantial runtime reductions compared to their CPU counterparts. In one dimension, the optimized kernels achieved notable speedups as the problem size increased, effectively offsetting the overhead of GPU memory transfers. In two dimensions, the benefits of GPU acceleration became even more pronounced: baseline implementations already reduced execution time significantly, and further optimization (e.g. memory coalescing and shared memory usage) yielded additional gains. At large problem sizes, the GPU consistently outperformed the CPU by more than an order of magnitude. Importantly, these performance improvements were achieved without compromising numerical accuracy, as evidenced by the convergence and error analysis of the optimized solver.

Taken together, the results highlight the *synergy between numerical analysis and high–performance computing.* Theoretical properties such as convergence, accuracy, and stability provide a rigorous foundation, while GPU acceleration enables these

computations to be carried out at scales relevant for scientific and engineering applications. The study illustrates that the transition from mathematical formulation to efficient large–scale simulation requires both careful algorithmic design and attention to architectural details of modern hardware.

In summary, the chapter demonstrates that finite difference solvers for hyperbolic PDEs can be both **mathematically robust** and **computationally efficient** when coupled with GPU acceleration. These findings not only validate the correctness of the developed CUDA solvers, but also provide a strong motivation for further exploration of GPU–based methods in the broader context of numerical simulation of wave phenomena.

Chapter 6

Conclusion

This thesis investigated the numerical solution of the two–dimensional wave equation using finite difference methods, with a particular focus on accuracy, stability, and performance optimization through GPU acceleration. The main findings can be summarized as follows.

From the perspective of *numerical correctness*, the implemented schemes demonstrated second–order convergence in both space and time. Convergence studies in one– and two–dimensional test problems confirmed the theoretical order of accuracy, while comparisons with exact solutions validated the reliability of the solvers. The stability analysis, supported by numerical experiments, confirmed the role of the CFL condition in ensuring boundedness of the numerical solution. These results collectively verified the mathematical soundness of the chosen schemes.

From the perspective of *computational performance*, GPU implementations offered significant speedups compared to CPU solvers, particularly for large problem sizes. In one dimension, CUDA kernels already demonstrated meaningful runtime reductions, while in two dimensions, the benefits were even more pronounced. Optimized CUDA kernels further improved memory efficiency and scalability, yielding performance improvements of more than an order of magnitude without any loss of accuracy. These results highlight the suitability of GPUs for large–scale time–dependent PDE simulations.

Taken together, this work demonstrates that combining rigorous **numerical analysis** with **high–performance GPU implementations** provides a powerful framework for solving hyperbolic PDEs. The study shows that well–understood mathematical schemes can be translated into efficient large–scale solvers when carefully adapted to the underlying hardware architecture.

Outlook

While the present work provides a solid foundation for the numerical simulation of wave propagation using finite difference methods and GPU acceleration, several directions for future research and development can be identified.

First, from a *numerical perspective*, the implemented schemes could be extended towards higher–order methods, such as discontinuous Galerkin or spectral element methods. These approaches may provide superior accuracy for smooth solutions and better dispersion properties for long–time wave propagation, albeit at the cost of increased algorithmic complexity. Incorporating adaptive mesh refinement (AMR) would also allow for more efficient resolution of localized features, such as sharp fronts or interfaces.

Second, from a *computational perspective*, the present CUDA implementation could be scaled further by introducing multi–GPU parallelism, or by combining CUDA with message passing (MPI) for distributed memory systems. This would make the solver applicable to massively parallel supercomputers, enabling simulations on domains of practical scientific and engineering interest. Additional performance optimizations, such as kernel fusion, asynchronous communication, or mixed precision arithmetic, could further enhance computational efficiency.

Third, from an *application perspective*, the solver could be extended and validated for more complex physical models. Potential directions include elastic wave propagation in heterogeneous media, wave scattering problems, or coupled fluid–structure interaction. Such extensions are particularly relevant for real–world applications in areas such as medical ultrasound imaging, seismic exploration, and aeroacoustics. Demonstrating the solver's applicability to these contexts would significantly broaden its impact.

Finally, the integration of the numerical solver into modern workflows, for example by combining it with machine learning–based surrogate models or uncertainty quantification frameworks, could open new research avenues. Hybrid approaches of this kind would combine the reliability of physics–based simulation with the adaptability of data–driven methods.

In conclusion, this thesis represents a step towards bridging the gap between rigorous numerical methods and efficient large–scale implementation on modern hardware. The outlook outlined above suggests a range of opportunities for both methodological advances and practical applications, continuing the development towards robust and high–performance wave simulation tools.

Bibliography

Richard Courant, Kurt Friedrichs, and Hans Lewy. Über die partiellen differenzengleichungen der mathematischen physik. *Mathematische Annalen*, 100(1):32–74, 1928.

Alexandre Ern and Pascal Steins. Convergence and superconvergence of the leapfrog scheme for second-order evolution equations. *Mathematics of Computation*, 89 (324):1717–1744, 2020.

Marcus J. Grote and Tsvetelin Mitkova. Explicit local time-stepping methods for the wave equation. *Journal of Computational Physics*, 403:109064, 2020.

Peter D. Lax and Robert D. Richtmyer. Survey of the stability of linear finite difference equations. *Communications on Pure and Applied Mathematics*, 9(2): 267–293, 1956.

Randall J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. SIAM, 2007.

Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2021.

Gilbert Strang. *Computational Science and Engineering*. Wellesley-Cambridge Press, 2007.

John C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations*. SIAM, 2004.

Lloyd N. Trefethen. *Finite Difference and Spectral Methods for Ordinary and Partial Differential Equations*. Unpublished Textbook, available online, 1996. `https://people.maths.ox.ac.uk/trefethen/fds.pdf`.